**day25\_107856406\_dsdipt\_sudipto\_7august2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 7 August 2025 (Day 25)

### ***Task 00 : Types of Design Patterns***

**1. Creational Patterns → *How objects are created***

These patterns abstract the instantiation process.

* **Singleton**
  + Ensures **only one instance** of a class exists.
  + Used in configs, logging, DB connections.
  + Private constructor, static instance, global access.
* **Factory Method**
  + Subclasses decide which class to instantiate.
  + More **flexible object creation**.
  + Ideal when the exact type isn't known in advance.
* **Abstract Factory**
  + Creates **families of related objects** without specifying their concrete classes.
  + Used in UI toolkits: Button + ScrollBar families for Mac/Windows.
* **Builder**
  + Step-by-step object construction.
  + Useful when objects are **complex and have multiple optional fields**.
* **Prototype**
  + Clones existing objects instead of creating new ones from scratch.
  + Great for performance-heavy creation scenarios.

**2. Structural Patterns → *How objects are composed/structured***

It helps us build relationships between entities.

* **Adapter (Wrapper)**
  + Converts one interface to another.
  + Like a charger adapter: lets incompatible interfaces work together.
* **Bridge**
  + Splits abstraction from implementation.
  + Lets you vary them independently — like themes + platforms.
* **Composite**
  + Tree structure for hierarchical data (folders within folders).
  + Treat **individuals and groups** uniformly.
* **Decorator**
  + Add new responsibilities to objects dynamically.
  + Like adding filters on a photo 📸.
* **Facade**
  + Simplified interface to a complex subsystem.
  + Like a hotel concierge: one interface for multiple services.
* **Flyweight**
  + Reduces memory usage by sharing data.
  + Good for large-scale object creation (like game sprites).
* **Proxy**
  + A placeholder that controls access to another object.
  + Used in lazy loading, access control, logging.

**3. Behavioral Patterns → *How objects interact/communicate***

Focus on communication patterns between objects.

* **Observer**
  + One-to-many dependency.
  + Like Instagram notifications: followers get updates when you post.
* **Strategy**
  + Defines a family of algorithms; encapsulates them.
  + Useful when you need multiple ways to do something (e.g., sort by name/date/price).
* **Command**
  + Encapsulates a request as an object.
  + Useful for **undo/redo**, or command queues.
* **Chain of Responsibility**
  + A request passes through a chain until handled.
  + Like tech support escalation: L1 → L2 → L3.
* **Mediator**
  + Centralized communication between objects.
  + Prevents spaghetti dependencies between colleagues (components).
* **State**
  + Object behavior changes with its internal state.
  + Like a traffic light (red/yellow/green states).
* **Template Method**
  + Defines the skeleton **of an algorithm**, lets subclasses tweak parts.
  + Used when steps are common but behavior varies.
* **Iterator**
  + Access elements of a collection without exposing its structure.
  + Think: .next() and .hasNext().
* **Visitor**
  + Separate algorithm from the object it operates on.
  + Allows adding new behavior without changing classes.
* **Interpreter**
  + Defines a grammar and an interpreter to interpret sentences in that grammar.
  + Used in SQL engines, expression evaluators, etc.

### ***Task 01 : Implement Builder Design Pattern***

**class User** {

// required

private String name;

private String email;

// optional

private int age;

private String phone;

// Private constructor — only Builder can access

private User(UserBuilder builder) {

this.name = builder.name;

this.email = builder.email;

this.age = builder.age;

this.phone = builder.phone;

}

// Static nested Builder class

public static class UserBuilder {

// required

private String name;

private String email;

// optional

private int age;

private String phone;

// Constructor with required fields

public UserBuilder(String name, String email) {

this.name = name;

this.email = email;

}

// Optional setters

public UserBuilder setAge(int age) {

this.age = age;

return this;

}

public UserBuilder setPhone(String phone) {

this.phone = phone;

return this;

}

// Final step - build the actual object

public User build() {

return new User(this);

}

}

// For displaying the object

public void display() {

System.out.println("Name: " + name);

System.out.println("Email: " + email);

System.out.println("Age: " + age);

System.out.println("Phone: " + phone);

}

}

**public class BuilderDemo** {

public static void main(String[] args) {

User user = new User.UserBuilder("eZcoDiN", "ez@mail.com")

.setAge(21)

.setPhone("9876543210")

.build();

user.display();

}

}

### ***Task 02 : Implement Adapter Design Pattern***

**interface TypeACable** {

void connectWithTypeA();

}

**class TypeCDevice** {

public void connectWithTypeC() {

System.out.println("Type-C Device: Connected with Type-C cable.");

}

}

**class CableAdapter implements TypeACable** {

private TypeCDevice typeCDevice;

public CableAdapter(TypeCDevice typeCDevice) {

this.typeCDevice = typeCDevice;

}

@Override

public void connectWithTypeA() {

System.out.println("Adapter: Converting Type-A to Type-C...");

typeCDevice.connectWithTypeC();

}

}

**public class AdapterDemo** {

public static void main(String[] args) {

// Type-C device (Adaptee)

TypeCDevice typeCDevice = new TypeCDevice();

// client expects a Type-A cable (Target interface)

TypeACable cable = new CableAdapter(typeCDevice);

// use an adapter to connect

System.out.println("Client: Connecting using Type-A interface...");

cable.connectWithTypeA();

}

}

### ***Task 03 : Implement Bridge Design Pattern***

Abstraction → Device

Implementor → Brand

ConcreteImplementor → Apple, Samsung

RefinedAbstraction → SmartPhone, Tablet

* **Decouples logic** – device logic & brand logic evolve separately.
* **Scalability** – Add new devices or new brands easily.
* **Avoids explosion of classes** – Without this, we would have to make: AppleSmartPhone, AppleTablet, SamsungSmartPhone, SamsungTablet…

**interface Brand** {

void powerOn();

void powerOff();

}

**class Apple implements Brand** {

public void powerOn() {

System.out.println("Apple device powering ON");

}

public void powerOff() {

System.out.println("Apple device powering OFF");

}

}

**class Samsung implements Brand** {

public void powerOn() {

System.out.println("Samsung device powering ON");

}

public void powerOff() {

System.out.println("Samsung device powering OFF");

}

}

**abstract class Device** {

protected Brand brand;

public Device(Brand brand) {

this.brand = brand;

}

abstract void turnOn();

abstract void turnOff();

}

**class SmartPhone extends Device** {

public SmartPhone(Brand brand) {

super(brand);

}

public void turnOn() {

System.out.print("ON SmartPhone: ");

brand.powerOn();

}

public void turnOff() {

System.out.print("OFF SmartPhone: ");

brand.powerOff();

}

}

**class Tablet extends Device** {

public Tablet(Brand brand) {

super(brand);

}

public void turnOn() {

System.out.print("ON Tablet: ");

brand.powerOn();

}

public void turnOff() {

System.out.print("OFF Tablet: ");

brand.powerOff();

}

}

**public class BridgeDemo** {

public static void main(String[] args) {

Device myPhone = new SmartPhone(new Apple());

myPhone.turnOn();

myPhone.turnOff();

System.out.println("----------------------------");

Device myTablet = new Tablet(new Samsung());

myTablet.turnOn();

myTablet.turnOff();

}

}

### ***Task 04 : Implement Composite Method Design Pattern***

It’s used when we want to treat **individual objects** and **compositions of objects** *uniformly*.

* A Manager can have subordinates (both Developers or other Managers)
* A Developer is a leaf node (can’t have subordinates)
* But we treat all employees the same via Employee interface.

**How It Works:**

* Both Developer and Manager implement Employee
* We can add Developers into Manager
* We can call showDetails() on any Employee — leaf or composite both!

**public interface Employee** {

void showDetails();

}

**public class Developer implements Employee** {

private String name;

private String role;

public Developer(String name, String role) {

this.name = name;

this.role = role;

}

@Override

public void showDetails() {

System.out.println("Developer: " + name + " - " + role);

}

}

**public class Manager implements Employee** {

private String name;

private String department;

private List<Employee> team;

public Manager(String name, String department) {

this.name = name;

this.department = department;

this.team = new ArrayList<>();

}

public void addEmp(Employee emp) {

team.add(emp);

}

public void removeEmp(Employee emp) {

team.remove(emp);

}

@Override

public void showDetails() {

System.out.println("Manager: " + name + " - " + department);

for (Employee e : team) {

e.showDetails();

}

}

}

**public class CompositeDemo** {

public static void main(String[] args) {

// Leaf nodes

Employee dev1 = new Developer("Alice", "Java Developer");

Employee dev2 = new Developer("Bob", "Python Developer");

// Composite node

Manager engManager = new Manager("Carol", "Engineering");

// Add developers under manager

engManager.addEmp(dev1);

engManager.addEmp(dev2);

engManager.showDetails();

dev1.showDetails();

}

}

### ***Task 05 : Implement Decorator Method Design Pattern***

The **Decorator Pattern** lets you **dynamically add behavior** to an object *without changing its original class*. Think of it like adding filters or toppings without touching the base dish.

Suppose we are ordering a coffee. The base is plain black coffee. We can add:

* Milk
* Sugar
* Ice

Each topping "decorates" the base drink without modifying the coffee class.

**public interface Beverage** {

String getDescription();

double getCost();

}

**public class Coffee implements Beverage** {

@Override

public String getDescription() {

return "Black Coffee";

}

@Override

public double getCost() {

return 50.0;

}

}

**public abstract class BeverageDecorator implements Beverage** {

protected Beverage beverage;

public BeverageDecorator(Beverage beverage) {

this.beverage = beverage;

}

}

**public class MilkDecorator extends BeverageDecorator** {

public MilkDecorator(Beverage beverage) {

super(beverage);

}

@Override

public String getDescription() {

return beverage.getDescription() + ", Milk";

}

@Override

public double getCost() {

return beverage.getCost() + 10.0;

}

}

**public class SugarDecorator extends BeverageDecorator** {

public SugarDecorator(Beverage beverage) {

super(beverage);

}

@Override

public String getDescription() {

return beverage.getDescription() + ", Sugar";

}

@Override

public double getCost() {

return beverage.getCost() + 5.0;

}

}

**public class DecoratorDemo** {

public static void main(String[] args) {

// Base coffee

Beverage myCoffee = new Coffee();

System.out.println("Order Plain Coffee: " + myCoffee.getDescription());

System.out.println("Total: ₹" + myCoffee.getCost());

// Add milk

myCoffee = new MilkDecorator(myCoffee);

System.out.println("Order Coffee with Milk: " + myCoffee.getDescription());

System.out.println("Total: ₹" + myCoffee.getCost());

// Add sugar

myCoffee = new SugarDecorator(myCoffee);

System.out.println("Order Coffee with Milk and Sugar: " + myCoffee.getDescription());

System.out.println("Total: ₹" + myCoffee.getCost());

}

}

### ***Task 06 : Implement Facade Method Design Pattern***

The **Facade Pattern** provides a **simplified interface** to a **complex subsystem**.

* Simplifying library usage
* Hiding multiple system calls behind a single method
* Making legacy code accessible in a modern way

**public class CPU** {

public void freeze() {

System.out.println("Freezing processor...");

}

public void execute() {

System.out.println("Executing instructions...");

}

}

**public class Memory** {

public void load(long position, String data) {

System.out.println("Loading data to memory: " + data + " at position " + position);

}

}

**public class Disk** {

public String read(long lba, int size) {

String data = "OS Boot Data";

System.out.println("Reading " + size + " bytes from disk at position " + lba);

return data;

}

}

**public class ComputerFacade** {

private CPU cpu;

private Memory memory;

private Disk disk;

public ComputerFacade() {

this.cpu = new CPU();

this.memory = new Memory();

this.disk = new Disk();

}

public void startComputer() {

System.out.println("----- Booting System -----");

cpu.freeze();

String data = disk.read(100, 512);

memory.load(100, data);

cpu.execute();

System.out.println("----- System Ready -----");

}

}

**public class FacadeDemo** {

public static void main(String[] args) {

ComputerFacade myPC = new ComputerFacade();

myPC.startComputer(); // one method call to start whole system!

}

}

### ***Task 07 : Implement Flyweight Method Design Pattern***

Flyweight is all about minimizing memory usage by sharing as much data as possible with similar objects.

**Use case:**

* When we have tons of similar objects.
* Instead of duplicating, we **reuse shared intrinsic data** and keep unique data separate.

**class Tree** {

private int x;

private int y;

private TreeType type;

public Tree(int x, int y, TreeType type) {

this.x = x;

this.y = y;

this.type = type;

}

public void draw() {

type.draw(x, y);

}

}

**class TreeType** {

private String name;

private String color;

private String texture;

public TreeType(String name, String color, String texture) {

this.name = name;

this.color = color;

this.texture = texture;

}

public void draw(int x, int y) {

System.out.println("Drawing " + name + " tree at (" + x + ", " + y + ")");

}

}

**class TreeFactory** {

private static final HashMap<String, TreeType> treeTypes = new HashMap<>();

public static TreeType getTreeType(String name, String color, String texture) {

String key = name + "\_" + color + "\_" + texture;

if (!treeTypes.containsKey(key)) {

treeTypes.put(key, new TreeType(name, color, texture));

System.out.println("Creating new TreeType: " + key);

}

return treeTypes.get(key);

}

}

**public class FlyweightDemo** {

public static class Forest {

private List<Tree> trees = new ArrayList<>();

public void plantTree(int x, int y, String name, String color, String texture) {

TreeType type = TreeFactory.getTreeType(name, color, texture);

Tree tree = new Tree(x, y, type);

trees.add(tree);

}

public void drawForest() {

for (Tree tree : trees) {

tree.draw();

}

}

public static void main(String[] args) {

Forest forest = new Forest();

forest.plantTree(1, 1, "Oak", "Green", "Rough");

forest.plantTree(2, 3, "Oak", "Green", "Rough");

forest.plantTree(5, 7, "Pine", "Dark Green", "Smooth");

forest.plantTree(9, 6, "Oak", "Green", "Rough"); // reused

forest.drawForest();

}

}

}

### ***Task 08 : Implement Proxy Method Design Pattern***

The **Proxy Pattern** is like a object.

It controls access — sometimes for security, logging, lazy loading, or even cost-saving reasons.

**public interface Internet** {

void connectTo(String serverHost) throws Exception;

}

**public class RealInternet implements Internet** {

@Override

public void connectTo(String serverHost) {

System.out.println("Connecting to " + serverHost);

}

}

**public class InternetProxy implements Internet** {

private Internet realInternet = new RealInternet();

private static final List<String> bannedSites = Arrays.asList("facebook.com", "instagram.com", "tiktok.com");

@Override

public void connectTo(String serverHost) throws Exception {

if (bannedSites.contains(serverHost.toLowerCase())) {

throw new Exception("Access Denied to " + serverHost);

}

realInternet.connectTo(serverHost);

}

}

**public class ProxyDemo** {

public static void main(String[] args) {

Internet internet = new InternetProxy();

try {

internet.connectTo("geeksforgeeks.org");

internet.connectTo("tiktok.com"); // This should get blocked

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

### ***Task 09 : Create a class diagram in UML***

@startuml

' Base interface

interface Coffee {

+ getDescription(): String

+ getCost(): double

}

' Concrete component

class SimpleCoffee implements Coffee {

+ getDescription(): String

+ getCost(): double

}

' Abstract decorator

abstract class CoffeeDecorator implements Coffee {

- decoratedCoffee: Coffee

+ getDescription(): String

+ getCost(): double

}

' Concrete decorators

class Milk extends CoffeeDecorator {

+ getDescription(): String

+ getCost(): double

}

class Sugar extends CoffeeDecorator {

+ getDescription(): String

+ getCost(): double

}

' Relationships

Coffee <|.. SimpleCoffee

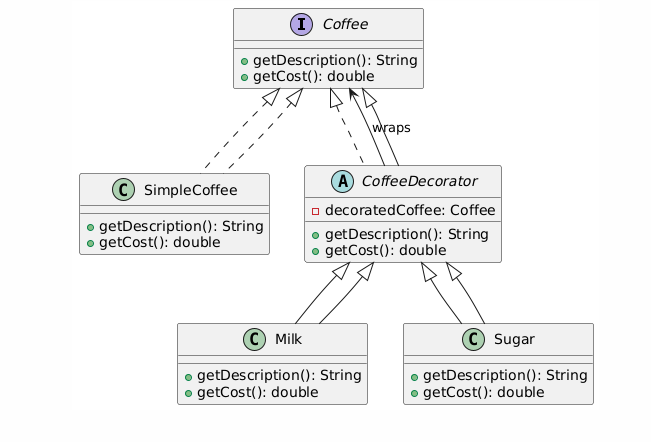
Coffee <|-- CoffeeDecorator

CoffeeDecorator <|-- Milk

CoffeeDecorator <|-- Sugar

CoffeeDecorator --> Coffee : wraps

@enduml



### ***Task 10 : What are the commonly used visibility Notations in class diagrams?***

@startuml

class Sample {

+ publicAttribute: int

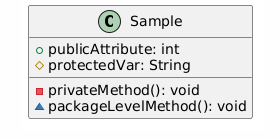
- privateMethod(): void

# protectedVar: String

~ packageLevelMethod(): void

}

@enduml



### ***Task 11 : What do you know about Parameter Directionality?***

@startuml

class Calculator {

+ calculateSum(in num1: int, in num2: int, out result: int)

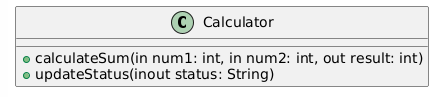
+ updateStatus(inout status: String)

}

@enduml

**This shows:**

* num1 and num2 are inputs only
* result is computed and sent back out
* status goes in and gets modified



### ***Task 11 : What do you know about Parameter Directionality?***

**1. Association (→)**

* **What:** A basic "has-a" relationship between two classes.
* **Direction:** Can be unidirectional or bidirectional.
* **Example:** Student → Course (Student *has a* Course)

class Student

class Course

Student --> Course : enrolls in

**2. Aggregation (◇→)**

* **What:** A **whole-part** relationship, but the part can exist independently.
* **Example:** Department ◇-- Professor (Professors can exist without the department)

class Department

class Professor

Department o-- Professor : has

**3. Composition (◆→)**

* **What:** A **stronger** form of aggregation. If the whole is destroyed, parts go too.
* **Example:** House ◆-- Room (If House is destroyed, so are its Rooms)

class House

class Room

House \*-- Room : contains

**4. Generalization / Inheritance ( ↖, shown as —|> or ^)**

* **What:** Classic OOP inheritance — "is-a" relationship.
* **Example:** Dog —|> Animal (Dog *is an* Animal)

class Animal

class Dog

Dog --|> Animal

**5. Realization (—|..)**

* **What:** A class **implements** an interface.
* **Example:** Car —|.. Drivable

interface Drivable

class Car

Car ..|> Drivable

**6. Dependency (..>)**

* **What:** One class **uses** another temporarily.
* **Example:** A method param or local variable.

class Order

class Logger

Order ..> Logger : uses temporarily

### ***Task 12 : UML Package Notation***

A package is a grouping of related classes, interfaces, sub-packages, or other UML elements. It helps in:

* Structuring large systems
* Managing dependencies
* Reflecting namespaces or modules in code

@startuml

package "User Module" {

class User

class Profile

}

package "Auth Module" {

class LoginService

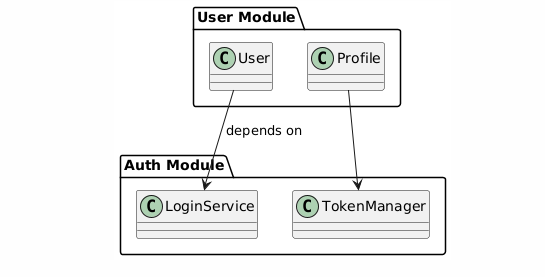
class TokenManager

}

User --> LoginService : depends on

Profile --> TokenManager

@enduml



### ***Task 13 : What is State Notation in UML?***

**🔹 Initial State**

* **Symbol:** A **filled black circle** ⚫
* **Meaning:** Where the object **starts** its lifecycle
* **UML keyword:** [\*] in PlantUML

[\*] --> Idle

**🔹 Final State**

* **Symbol:** A **bullseye** 🎯 (black circle with an outer ring)
* **Meaning:** The **end** of the object’s lifecycle or process
* **UML keyword:** [\*] --> when going to final, or [Final]

Running --> [\*]

@startuml

title Coffee Machine State Machine

[\*] --> Idle

Idle --> Brewing : press start

Brewing --> Dispensing : timer expires

Dispensing --> Idle : done

Idle --> [\*] : shutdown

@enduml

